Accelerated AC Contingency Calculation on Commodity Multi-core SIMD CPUs

Tao Cui, *Student Member, IEEE,* Rui Yang, *Student Member, IEEE,* Gabriela Hug, *Member, IEEE*
Franz Franchetti, *Member, IEEE*

***Abstract*—Multi-core CPUs with multiple levels of parallelism (i.e. data level, instruction level and task/core level) have become the mainstream CPUs for commodity computing systems. Based on the multi-core CPUs, in this paper we developed a high performance computing framework for AC contingency calculation (ACCC) to fully utilize the computing power of commodity systems for online and real time applications. Using Woodbury matrix identity based compensation method, we transform and pack multiple contingency cases of different outages into a fine grained vectorized data parallel programming model. We implement the data parallel programming model using SIMD instruction extension on x86 CPUs, therefore, fully taking advantages of a CPU core with SIMD floating point capability. We also implement a thread pool scheduler for ACCC on multi-core CPUs which automatically balances the computing loads across CPU cores to fully utilize the multi-core capability. We test the ACCC solver on the IEEE test systems and on the Polish 3000-bus system using a quad-core Intel Sandy Bridge CPU. The optimized ACCC solver achieves close to linear speedup (SIMD width multiply core numbers) comparing to scalar implementation and is able to solve a complete N-1 line outage AC contingency calculation of the Polish grid within one second on a commodity CPU. It enables the complete ACCC as a real-time application on commodity computing systems.**

I. Introduction

AC contingency calculation (ACCC) is the fundamental tool for power system steady state security assessment. It evaluates the consequences of power grid component failures, assesses the system security given the failures and further helps to provide corrective or preventive actions for decision making. ACCC is a basic module for most offline power system planning tools [1]. It is also a critical functionality of most SCADA/EMS systems. ACCC has also been widely used in market applications, such as simultaneous feasibility test for a market dispatch's security [2]. A complete ACCC computation on practical a power grid often requires a large amount of load flow computation, due to the computational burden, the complete ACCC often remains as offline / non real-time applications on commodity computing systems.

Recent large scale integration of variable renewable generation and large variance load (such as electric vehicle charging) introduce significant uncertainties and largely-varying grid conditions. Moreover, the increasing loads and generations drive today's power grid closer to its limits, resulting in higher possibility of contingency as well as more serious consequences of the contingency. The largely-varying grid conditions on already stressed power grid requires merging of most conventional offline analyses into online even real-time operation to satisfy the unprecedented security requirements. Therefore, an optimized fast ACCC solver for online and real-time operation would be an important tool given the new security assessment challenges.

This work was supported by NSF through awards 0931978 and 1116802.

In the computing industry, the performance capability of the computing platform has been growing rapidly in last several decades at a roughly exponential rate [3] [4]. The mainstream commodity CPUs enable us to build inexpensive systems with similar computational power comparing to supercomputers just ten years ago. However, these advances in hardware performance result from the increasing complexity of the computer architecture, thus they increase the difficulty of fully utilizing the available computational power for a specific application [4] [5]. In this paper, our work focuses on *fully* utilizing the computing power of modern CPU by code optimization and parallelization for specific hardware, enabling real-time complete ACCC for practical power grids on commodity computing system.

Related work. Contingency analysis has long since been a joint research field of both power system and high performance computing domains [6] [7]. In [8], a workload balancing method is developed for massive contingency calculation on Cray supercomputer. In [9], a hybrid approach is proposed using Cray XMT's graphical processing capability. A graphical processing unit based DC contingency analysis has been implemented in [10]. Recently, some commercial packages such as PSS/E also work actively to include parallel processing on multi-core CPU to boost the performance of ACCC [1]. However, most previous approaches are focusing on task level parallelism, or using specified parallel math solver. In order to fully utilize the computing power of commodity CPU with multi-level parallel hardware and other performance enhancement features, a specific algorithmic transform with code optimization for ACCC is presented in this paper.

Contribution. In this paper, we presented an accelerated ACCC that builds upon several algorithmic and computer architectural optimizations. At data level, we use Woodbury matrix identity with fast decoupled power flow algorithm to formulate a fine grain vectorized implementation of ACCC. It solves multiple cases simultaneously using SIMD (Single Instruction Multiple Data) floating point units of the modern CPU core. At the task/core level, we implement a thread pool scheduler on multi-core CPUs that automatically balances the computing loads across CPU cores. Together with some other aggressive code optimizations for data structure, sparse kernels and instruction level parallelism, as a result, our solver is able to complete a full line outages ACCC of a Polish 3120-bus system within a second. It enables real-time complete ACCC for a practical grid on inexpensive computing system.

Synopsis. The paper is organized as follows: the feature of computing platforms are reviewed in Section II. The SIMD transformation of ACCC is described in Section III. The multi-core and other optimizations are in Section IV. We report the performance results of the optimized solver in Section V. Section VI concludes the paper.

II. Modern Computing Platform

Fig. 1 shows the structure of the quadcore Intel Core i7 2670 QM Sandy Bridge CPU for mid-range laptops. It has 4 physical cores (Core P#0-4), three levels (L1-L3) of CPU cache memory. It also supports Intel's new AVX (Advanced Vector eXtension) instruction set for single instruction multiple data (SIMD) operations. It has a clock rate of 2.2 GHz. The theoretical single precision peak performance is 140 Gflop/s (or Gflops, 10^9 floating point operations per second) [11]. In terms of this value, this performance oriented CPU in 2012 has the similar performance as the fastest supercomputer in the world in just year 2001 (Cray T3E1200, 138 Gflop/s on Top500 List) [3]. However, the peak hardware performance assumes that one can *fully* utilize all the performance enhancement features of the CPU, which becomes very difficult on modern CPUs given the more and more complicated hardware. We mainly look into the following hardware aspects explicitly available to software development:

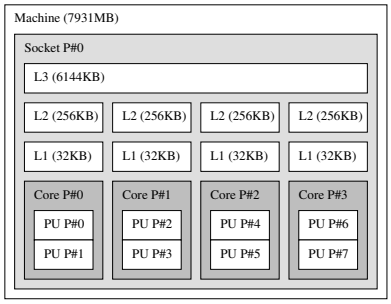

Fig. 1. Core i7 2670QM CPU system structure: 4-core, 3-level cache

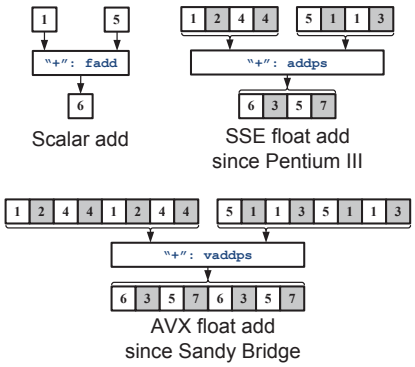

Fig. 2. Illustration of scalar add versus SIMD add

Multiple levels of parallelism have become the major driving force for the hardware performance. The following are two explicit parallelism on modern CPU:

1) Single Instruction Multiple Data (SIMD) uses special instructions and registers to perform same operation on multiple data at the same time: The Streaming SIMD Extensions (SSE) or the new Advanced Vector eXtensions (AVX) instruction sets on Intel or AMD CPUs perform floating point operations on 4 floating point or 8 floating point data packed in vector register at the same time (single precision). As illustrated in Fig. 2 for example, the scalar `fadd` performs add operation on one data slot, the SSE version `addps` or AVX version `vaddps` instruction performs the add operation on four or eight data slots simultaneously. Many new or under-developing micro-architectures such as Intel's new Xeon Phi architecture further expands the SIMD processing width to 16 data. With the help of SIMD, the performance can be significantly increased for particular formed problems.

2) Multithreading on multi-core CPUs enables multiple threads to be executed simultaneously and independently on different CPU cores while communicate via shared memories. A proper scheduling and load balancing strategy is necessary to fully utilize multiple CPU cores.

Memory hierarchy, e.g. multiple levels of caches, should also be considered for performance tuning. The cache is a small but fast memory that automatically keeps and manages copies of the most recently used and the most adjacent data from the main memory locations in order to bridge the speed gap between fast processor and slow main memories. There could be multiple levels of caches (L1, L2, L3 in Fig. 1), the levels closer to CPU cores are faster in speed but smaller in size. An optimized data storage and access pattern is important to utilize the caches functions to increase the performance, instead of the slow memories.

Given the hardware features, proper parallelization model and code optimization techniques are of crucial importance to *fully* utilize the computing power for ACCC performance. In the following sections, we show programming model, code optimization methods, and the benefit we can obtain by applying performance tuning and parallel programming model on modern multi-core hardware platforms for ACCC.

III. Mapping Contingencies to Data Parallelism

The basic computation of contingency calculation is solving multiple cases of power flow considering the component failure such as transmission line or generator outage.

A. Fast decoupled power flow formulation

Fast decoupled power flow (FDPF) is a widely used power flow solution for ACCC. FDPF is originated from full Newton-Raphson (NR) method, by considering some unique properties of transmission network [12]. FDPF method often has fewer total floating point operation comparing to NR method [13]. In each iteration of FDPF, following equations are solved one after another to update the state $(\theta, \mathbf{V})$. Suppose the state of current iteration is $(\theta^k, \mathbf{V}^{(k)})$, the state of next iteration $(\theta^{(k+1)}, \mathbf{V}^{(k+1)})$ is computed as following:

$$\begin{aligned} \mathbf{\Delta V}^{(k)} &= -\mathbf{B''}^{-1}\mathbf{\Delta Q}(\theta^{(k)}, \mathbf{V}^{(k)})/\mathbf{V}^{(k)} \\ \mathbf{V}^{(k+1)} &= \mathbf{V}^{(k)} + \mathbf{\Delta V}^{(k)} \end{aligned} \tag{1}$$

$$\begin{aligned} \mathbf{\Delta}\theta^{(k)} &= -\mathbf{B'}^{-1}\mathbf{\Delta P}(\theta^{(k)}, \mathbf{V}^{(k+1)})/\mathbf{V}^{(k+1)} \\ \theta^{(k+1)} &= \theta^{(k)} + \mathbf{\Delta}\theta^{(k)} \end{aligned} \tag{2}$$

In equation (2) and (1), $\mathbf{\Delta Q}(\cdot)$ and $\mathbf{\Delta P}(\cdot)$ compute the power mismatch by matrix-vector product styled operations.

Two linear systems of $\mathbf{B'}$ and $\mathbf{B''}$ (the inverse in above equations) are solved to obtain the adjustment $\Delta\theta$ and $\Delta\mathbf{V}$. Using direct linear solver, the $\mathbf{B'}$ and $\mathbf{B''}$ are pre-factorized into the product of lower triangle (L) and upper triangle (U) matrices before the iteration: $\mathbf{B'} = L'U'$,$\mathbf{B''} = L''U''$. During the iteration, only forward and backward substitutions using LU factors are needed to solve the linear systems.

B. Contingency calculation using FDPF

The core computation of ACCC is to solve multiple cases of AC power flow given different component failures. These contingency cases can be considered as a same power flow base case without failure, plus different modifications on the equation and/or parameters to consider contingencies. The ACCC can be formulated as modified base case load flow solutions. Following shows the typical modifications:

Line outage cases. In these cases, the system parameters keeps unchanged, suppose the failed line section from bus i to bus j is taken out of the system. As a result, a 2×2 matrix Δy is added to corresponding slot of original admittance matrix Y to form the new admittance matrix $\tilde{Y}$.

$$M = \begin{bmatrix} 0, ..., \underset{i}{1}, 0...0, 0, ..., 0 \\ 0, ..., 0, 0...0, \underset{j}{1}, ..., 0 \end{bmatrix}^T \tag{3}$$

$$\tilde{Y} = Y + M\Delta y M^T \tag{4}$$

$$\Delta y = \begin{bmatrix} y_{ij} + b_{ij} & -y_{ij} \\ -y_{ij} & y_{ij} + b_{ij} \end{bmatrix} \tag{5}$$

In FDPF computation, this will affect the mismatch using the admittance matrix in the matrix-vector product and the linear solver using modified $\mathbf{B}'$ and $\mathbf{B}''$:

$$\tilde{\mathbf{B}}' = \mathbf{B}' + M'\Delta b' M'^T \tag{6}$$

$$\tilde{\mathbf{B}}'' = \mathbf{B}'' + M''\Delta b'' M''^T \tag{7}$$

PV bus outage cases. Such case happens when the generator fails to maintain the voltage of a PV bus. The PV bus changes to PQ bus. The admittance matrix Y does not change. While in the iteration steps (1), another equation for new PQ bus' Q and the unknown voltage will be added. Resulting another column and row added to the bottom and right of B'':

$$\tilde{\mathbf{B}}'' = \begin{bmatrix} \mathbf{B}'' & N \\ N^T & a \end{bmatrix} \tag{8}$$

Generator outage without PV bus outage cases. These cases only affect the specified active power injection on PV bus, without modifying the power flow equation system.

C. Data parallelism using compensation method

Transform ACCC into data level parallelism is based on the *Woodbury matrix identity* in matrix theory or *Compensation method* in circuit simulation. Suppose:

$$\tilde{A} = A + MaN^T \tag{9}$$

The inverse of $\tilde{A}$ is

$$\tilde{A}^{-1} = A^{-1} - A^{-1}M(a^{-1} + N^T A^{-1} M)^{-1} N^T A^{-1} \tag{10}$$

Based on above formula, we can compute the contingency using the base case factorization result with minimal extra computation. Take $\mathbf{B}'$ as example, $\mathbf{B}'$ is pre-factorized as:

$$\mathbf{B}' = L'U' \tag{11}$$

Handling line outage: In the base case, we need to solve x for $\mathbf{B}'x = b$ in each iteration. While in the line outage cases, we need to solve x for:

$$(\mathbf{B}' + M'\Delta b' M'^T)x = b \tag{12}$$

Based on *Woodbury matrix identity*, the solution for (12) can be formulated as following steps:

Forward substitution:

$$F = L'^{-1} b \tag{13}$$

Compensate:

$$W = L'^{-1} M' \tag{14}$$

$$\tilde{W}^T = M'^T U'^{-1} \tag{15}$$

$$c = (\Delta b^{-1} + \tilde{W}^T W)^{-1} \tag{16}$$

$$\Delta F = -W c \tilde{W}^T F \tag{17}$$

$$F = F + \Delta F \tag{18}$$

Backward substitution:

$$\tilde{x} = U'^{-1} \tilde{F} \tag{19}$$

Note in compensation steps (14) to (17), the computation only related to the new system topology, therefore, these compensation matrices can be pre-computed before the ACCC. Also W and W^T which have same dimension as M' and M'^T, and with a proper ordering scheme, these two matrices can be sparse with small floating-point operation numbers and memory footprint.

$\mathbf{B}''$ can be treated in the similar way. Therefore, during the contingency computation (13) and (19) are fixed procedure for all cases. Different contingency cases can be computed using the compensation step based on pre-computed W, $\tilde{W}^T$, c and (17) (18). The mismatch is computed based on modified admittance matrix with same instruction different data.

Handling PV bus outage. The base case solving x for $\mathbf{B}''x = b$ becomes solve $\tilde{x}$ for:

$$\tilde{\mathbf{B}}''\tilde{x} = \begin{bmatrix} \mathbf{B}'' & N \\ N^T & a' \end{bmatrix} \tilde{x} = \begin{bmatrix} b \\ b' \end{bmatrix} \tag{20}$$

In the similar way, solving above equation can be decomposed into following steps:

Step 1: solving x_0 for

$$(\mathbf{B}'' - N(\frac{1}{a'})N^T)x_0 = b - (b'/a')N \tag{21}$$

Step 2: solve $\tilde{x}$:

$$x_1 = \frac{1}{a'}(b' - N^T x_0) \tag{22}$$

$$\tilde{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{23}$$

Note (21) can be solved in the same way using compensation as we solving (12) based on the same L'' and U'' factor of base case $\mathbf{B}''$ for different contingencies.

From above compensation method, the contingency cases can be decomposed into following types of operation:

0) Pre-computation for LU factors and compensation matrices for different contingencies
1) Fixed mismatch calculation using slightly changed admittance matrix
2) Fixed forward / backward substitution for all cases
3) Different compensation steps for different contingencies

The *fixed* computation steps in above list are mostly the same instruction sequence on different data. Only the compensation steps are different for different contingency cases

to accurately consider the contingencies' for the solution. With above decomposition of computing procedure, the *fixed* computation steps of the ACCC can be well mapped on to finer grain data level parallelism and can use SIMD instructions to performance computation on multiple cases simultaneously.

D. Programming model using SIMD instruction extensions

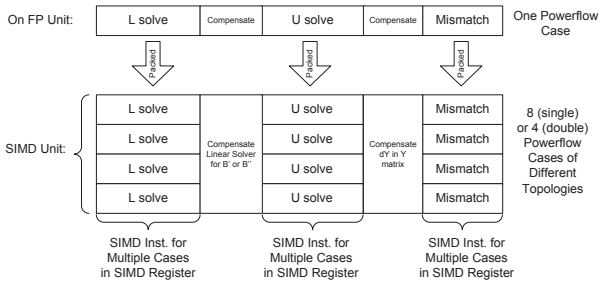

Fig. 3. SIMD implementation of ACCC: basic iteration segment

The SIMD model for different contingency calculation is showed in Fig. 3. The upper part of the figure shows the original scalar version code on CPU's floating-point unit: the contingency cases are evaluated sequentially. The lower part shows the SIMD version code using CPU's SIMD units: the forward/backward substitution parts of linear solver, the mismatch computation are vectorized and 4 cases (on SSE) or 8 cases (on AVX) are processed simultaneously on SIMD units, while the compensation for different cases are evaluated using pre-computed compensation matrices and then are plugged into the corresponding slots in SIMD units.

IV. Multi-core Task Scheduling and Others

A. Task scheduling over multiple cores

Load balancing is one of the most important considerations for designing parallel program. It is also one of the important target of most ACCC research projects and commercial products [1] [8]. In our ACCC application, we deal with the load balancing at the core level in a shared memory system: distributing and balancing the workload among multiple CPU cores to fully utilize the computing resources.

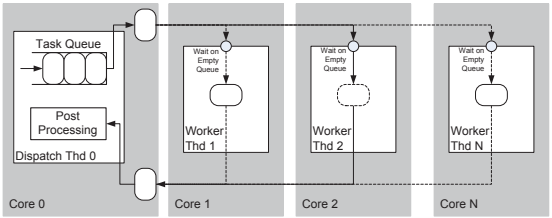

Fig. 4. Thread pool scheduler on multi-core CPU.

We implemented a thread pool based scheduler for the ACCC. As showed in Fig. 4, a pool of worker threads (Worker Thd 1 to N) are created and pinned to Core 1 to Core N to process the SIMD vectorized ACCC tasks. One dispatch thread (Dispatch Thd 0) is created and pinned to Core 0 to manage the task queue and dispatch the work tasks into the thread pool, as well as post process the ACCC results. The three elements of the thread pool scheduler design are the task queue data structure and the two types of threads:

1. Task queue data structure includes: a queue buffering the task pointers to the SIMD ACCC cases to be processed.
2. Worker thread: wait if the queue is empty, otherwise pop the task from the task queue to process using the SIMD ACCC solver in Fig. 3.
3. Dispatch thread: keep dispatches the task into the task queue, once all tasks are dispatched, wait on the queue status. When the queue is empty, finish and clean up.

In this way, whenever any worker thread finishes the tasks and the queue is not empty, the worker will get new task from the queue. In our ACCC application, there are usually a large amount of small tasks, the loads can be dynamically balanced among worker threads on different physical cores.

B. Other code optimization

Besides explicit parallelization, we applied the following code optimization techniques for load flow kernel on x86 CPU.

1. Sparse LU factorization using approximated minimal degree scheme (AMD) for solving $\mathbf{B}'$ and $\mathbf{B}''$ [14].
2. Optimized usage of trigonometric functions: using optimized trigonometric functions to achieve same precision with smaller number of CPU cycles [15].
3. Unrolling sparse kernel, pre-generate source code for consecutive columns of the sparse L and U factors to build bigger code block and using jump table to avoid branching in the inner loop. Similar approaches on unrolling techniques are discussed in [16] [17] [18].

V. Performance Results

In this section, we show the performance results of the accelerated ACCC solver.

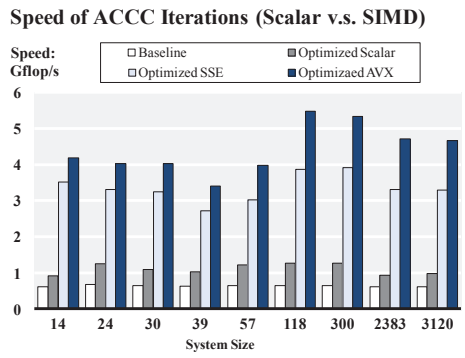

Fig. 5. Speedup result by SIMD transformation

Fig. 5 shows the performance breakdown of the data parallel implementation of our ACCC solver on a single CPU core for different test systems (including IEEE standard test systems from 14-bus to 300-bus and Polish grid of 2383 buses and 3120 buses). The performance results are showed in terms of Gflop/s. The base algorithm is the FDPF load flow algorithm with AMD based sparse LU factors. The lowest bar is the baseline implementations directly using sparse kernel from CXSparse package in SuiteSparse [19]. The second lowest bar is the optimized scalar implementation with the optimization techniques on sparse kernel and math functions discussed in

Section IV-B. Based on the optimized scalar implementation, the third bar shows the speed results of the SIMD implementation using SSE instruction extensions which are available on most x86 CPUs. Using SSE, our accelerated implementation processes packed 4 single precision floating point data at the same time, a close to linear speedup can be observed. The highest bar is the SIMD implementation using AVX instruction extensions available on Intel Sandy Bridge CPU since 2011. Using AVX, we pack 8 single precision floating point data and process the packed data using AVX instruction. Another speedup can be observed.

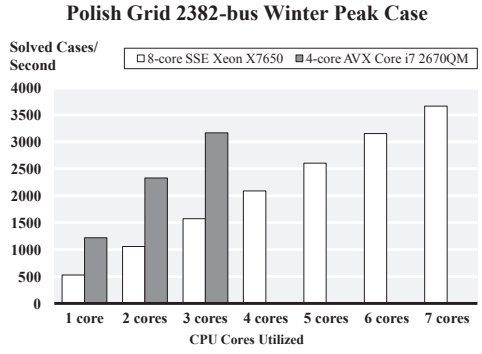

Fig. 6. Thread pool performance of Polish 2383-bus system

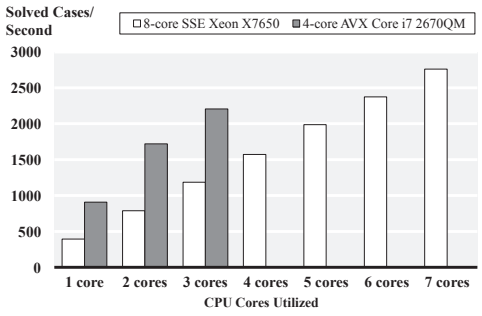

Fig. 7. Thread pool performance of Polish 3120-bus system

Fig. 6 and Fig. 7 shows the results in terms of how many contingency cases can be solved every second of Polish Grid. The maximal iteration is set to 20 and the maximal mismatch is 1kW. Most N-1 cases converged using FDPF. We show the test results on two machines, the darker bars are the results on a quad-core 2.2GHz Intel Core i7 2670QM Sandy Bridge CPU supporting AVX instructions. The lighter bars are the results of an 8-core 2.26GHz Intel Xeon X7560 Nehalem CPU supporting SSE 4.1 instructions. On each CPU, one thread (one core) is reserved for scheduler and post-processing, the rest cores are for load flow computations. In these tests on both machines, we observed a linear speedup for ACCC with the increased core numbers, thanks to the dynamic balance of the thread pool design. Also, the 4-core machine is able to achieve higher performance thanks to the AVX capability with wider SIMD processing width. In Fig. 6 and Fig. 7, our ACCC is able to finish a complete N-1 line outage screening for the Polish grid using each of these two CPUs around a second. Therefore, our implementation enables ACCC as a real-time application for the practical sized power grid to meet the new challenges of smart power grid.

VI. Conclusion

In this paper we presented a multi-core high performance accelerated ACCC solver. By applying various performance optimizations and multi-level parallelization, especially the compensation based algorithm transformation, we transform the ACCC into a fine grained data parallel model. We also implemented a thread pool scheduler that can dynamically balance the work loads. The proposed ACCC fully utilizes the computing capability of the modern CPUs and the performance is scalable with the hardware parallel capacity. We tested the ACCC solver on the IEEE test system as well as a real world national level Polish grid. Our ACCC solver is able to do a full N-1 line outage screening of Polish network within a second, enabling the fast ACCC solution for real-time operation on commodity computing systems.

References

[1] Siemens-PTI, "PSS/E: Power system simulator for engineering."
[2] PJM, "Auction revenue rights," http://www.pjm.com/about-pjm/learning-center/markets-and-operations/arrs.aspx.
[3] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 list," http://www.top500.org.
[4] J. Larus, "Spending Moore's dividend," *Commun. ACM*, vol. 52, pp. 62–69, May 2009.
[5] S. Chellappa, F. Franchetti, and M. Püschel, "How to write fast numerical code: A small introduction," *Generative and Transformational Techniques in Software Engineering II*, pp. 196–259, 2008.
[6] D. Falcão, "High performance computing in power system applications," *Vector and Parallel ProcessingVECPAR'96*, pp. 1–23, 1997.
[7] Z. Huang and J. Nieplocha, "Transforming power grid operations via high performance computing," in *Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, july 2008, pp. 1 –8.
[8] Z. Huang, Y. Chen, and J. Nieplocha, "Massive contingency analysis with high performance computing," in *Power Energy Society General Meeting, 2009. PES '09. IEEE*, july 2009, pp. 1 –8.
[9] I. Gorton, Z. Huang, Y. Chen, B. Kalahar, S. Jin, D. Chavarria-Miranda, D. Baxter, and J. Feo, "A high-performance hybrid computing approach to massive contingency analysis in the power grid," in *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, dec. 2009, pp. 277 –283.
[10] A. Gopal, D. Niebur, and S. Venkatasubramanian, "Dc power flow based contingency analysis using graphics processing units," in *Power Tech, 2007 IEEE Lausanne*, july 2007, pp. 731 –736.
[11] Intel Corporation, "Intel®microprocessor export compliance metrics," http://www.intel.com/support/processors/sb/cs-017346.htm.
[12] O. Alsac, B. Stott, and W. Tinney, "Sparsity-oriented compensation methods for modified network solutions," *Power Apparatus and Systems, IEEE Transactions on*, no. 5, pp. 1050–1060, 1983.
[13] B. Stott, "Review of load-flow calculation methods," *Proceedings of the IEEE*, vol. 62, no. 7, pp. 916 – 929, july 1974.
[14] P. Amestoy, T. Davis, and I. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 381–388, 2004.
[15] N. Shibata, "Efficient evaluation methods of elementary functions suitable for simd computation," *Computer Science-Research and Development*, vol. 25, no. 1-2, pp. 25–32, 2010.
[16] T. Cui and F. Franchetti, "Autotuning a random walk boolean satisfiability solver," *Procedia Computer Science*, vol. 4, pp. 2176–2185, 2011.
[17] ——, "A multi-core high performance computing framework for probabilistic solutions of distribution systems," in *Power and Energy Society General Meeting, 2012 IEEE*. IEEE, 2012, pp. 1–6.
[18] ——, "Optimized parallel distribution load flow solver on commodity multi-core cpu," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–6.
[19] T. Davis, I. Duff, P. Amestoy, J. Gilbert, S. Larimore, E. P. Natarajan, Y. Chen, W. Hager, and S. Rajamanickam, "SuiteSparse: a suite of sparse matrix packages."